Dynamic Programming

Similar to divide-and-conquer, but avoids duplicate work when subproblems are identical.

(Typically used for optimization problems like the Traveling Salesman Problem).

Matrix Multiplication

Problem: Find optimal parenthesization of a chain of matrices to be multiplied such that the number of scalar multiplications is minimized.

Recall matrix multiplication algorithm:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 10 \\ 8 & 11 \\ 9 & 12 \end{bmatrix} = \begin{bmatrix} 1*7+2*8+3*9 & 1*10+2*11+3*12 \\ 4*7+5*8+6*9 & 4*10+5*11+6*12 \end{bmatrix}$$

$$\text{2x3 * 3x2 = 2x2}$$

```
MatrixMultiply(A,B)
   for i = 1 to rows(A)
      for j = 1 to cols(B)
         C[i,j] = 0
         for k = 1 to cols(A)
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

$$A_{p*q}B_{q*r} = C_{p*r}$$

Thus the number of multiplications is p*q*r.

Matrix Multiplication Parenthesization

For example, $A_1A_2A_3$ can be rewritten as
$(A_1A_2)A_3$ or $A_1(A_2A_3)$.

Example

Suppose A_1 is 10x100, A_2 is 100x5, and A_3 is 5x50.

Then $A_1(A_2A_3) \longrightarrow$ 100*5*50 + 10*100*50 = 25,000 + 50,000 = ________ scalar multiplications (A_2A_3 is a 100x50 matrix).

$(A_1A_2)A_3 \longrightarrow$ 10*100*5 + 10*5*50 = 5,000 + 2,500 = _______ scalar multiplications (A_1A_2 is a 10x5 matrix).

Brute Force Solution: Try all possible parenthesizations

How many? ______

$A_1A_2...A_k \mid A_{k+1}...A_{n-1}A_n$
P(k)*P(n-k), k = 1 to (n-1)

$$P(n) = \begin{cases} 1 & n = 1 \\ \Sigma_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

See Cormen et al., Problem 13-4 for solving this recurrence.

$$\mathrm{P(n)} = \frac{1}{n}\begin{pmatrix} 2n-2 \\ n-1 \end{pmatrix}$$

$$= \Omega(\frac{4^{n-1}}{(n-1)^{\frac{3}{2}}}),\text{ which is exponential in n.}$$

Dynamic Programming Solution (4 steps)

1. Characterize the structure of an optimal solution.

2. Recursively define the value of an optimal solution.

3. Compute the value of an optimal solution in a bottom-up fashion.

4. Construct an optimal solution from computed information.

Step 1: Characterize Structure of Optimal Solution

Parenthesization of two subchains $A_1..A_k$ and $A_{k+1}..A_n$ must each be optimal for $A_1..A_n$ to be optimal.

Why? A lower cost solution to a subchain reduces the cost of $A_1..A_n$. The total cost is calculated as cost$(A_1..A_k)$ + cost$(A_{k+1}..A_n)$ + cost of multiplying two resultant matrices together. The last term is constant no matter what the subproblem solutions are.

We can show that if our subproblem solution is not optimal, a better subproblem solution cost yields a better total cost.

Thus, as is the case with ALL Dynamic Programming solutions, an optimal solution to the problem consists of optimal solutions to subproblems.

This is called ______________________________.

Step 2: Define recursive solution

Let $A_{i..j} = A_i A_{i+1}..A_j$, where A_i has dimensions P[i-1] x P[i]. P is an array of dimensions.

For now, the subproblems will be finding the minimum number of scalar multiplications m[i,j] for computing $A_{i..j}$ (1 ≤ i ≤ j ≤ n).

Define m[i,j].

- If i = j, m[i,j] = 0 (single matrix).
- If i < j, assume an optimal split between A_k and A_{k+1} (i ≤ k < j).
  m[i,j] = cost of computing $A_{i..k}$ + cost of computing $A_{k+1..j}$ + cost of computing $A_{i..k}A_{k+1..j}$
  = m[i,k] + m[k+1,j] + P[i-1]P[k]P[j]
  However, we do not know the value of k, so we have to try all ____ possibilities.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ min_{i \le k < j}(m[i,k] + m[k+1,j] + P[i-1]P[k]P[j] & \text{if } i < j \end{cases}$$

Note that a recursive algorithm based on this definition would still require exponential time.

Recursive Solution

Consider a recursive solution:

Let p = $< p_0, p_1, .., p_n >$ be the sequence of dimensions.

Recursive-Matrix-Chain(p,i,j)
  if i = j

then return 0
m[i,j] = ∞
for k = i to j - 1
    q = Recursive-Matrix-Chain(p,i,k) +
        Recursive-Matrix-Chain(p,k+1,j) +
        P[i-1]P[k]P[j]
    if q < m[i,j]
    then m[i,j] = q
return m[i,j]

Analysis:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + \sum_{k=1}^{n-1}(T(k) + T(n-k) + \Theta(1)) & n > 1 \end{cases}$$

$$\begin{aligned} T(n) &= \Theta(1) + \sum_{k=1}^{n-1}(T(k) + T(n-k) + \Theta(1)) \\ &= \Theta(1) + \sum_{k=1}^{n-1}\Theta(1) + \sum_{k=1}^{n-1}T(k) + \sum_{k=1}^{n-1}T(n-k) \\ &= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n-1}T(k) + \sum_{k=1}^{n-1}T(k) \\ &= \Theta(n) + 2\sum_{k=1}^{n-1}T(k) \end{aligned}$$

Analysis

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(n) + 2\sum_{k=1}^{n-1}T(k) & n > 1 \end{cases}$$

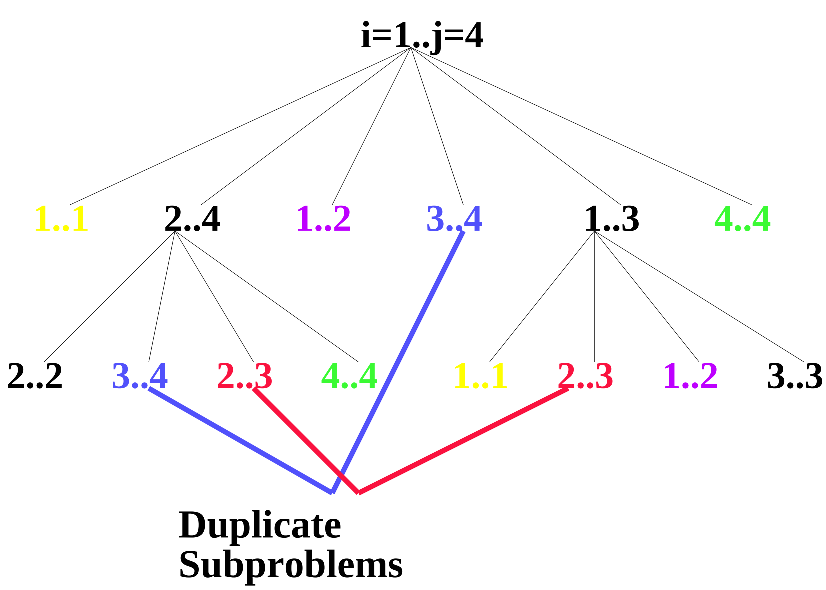
Want to show running time is at least exponential, so show T(n) = $\Omega(2^n)$.

By substitution method:
Show: T(n) $= \Omega(2^n) \geq c2^n$
Assume: T(k) $\geq c2^k$ for k $<$ n

$$
\begin{aligned}
T(n) &\geq \Theta(n) + 2\sum_{k=1}^{n-1} c2^k \\
&= \Theta(n) + 2c\sum_{k=0}^{n-2} 2^{k+1} \\
&= \Theta(n) + 4c\sum_{k=0}^{n-2} 2^{k} \\
&= \Theta(n) + 4c(2^{n-1} - 1) \\
&= \Theta(n) + 2c2^n - 4c \\
&\geq c2^n
\end{aligned}
$$

If $4c - \Theta(n) \leq 0$, or $c \leq \Theta(n)/4$ (okay for large enough n).
Thus, $T(n) = \Omega(2^n)$; still exponential.

Duplicate Subproblems



Unique Subproblems

How many unique subproblems?

Assume that $1 \leq i < j \leq n$ or $1 \leq i = j \leq n$.

$$\binom{n}{2} + n$$

All ways of choosing i and j for problem m[i,j] when i < j +
All ways of choosing i and j for problem m[i,j] when i = j

$$= \frac{n!}{2!(n-2)!} + n$$

$$= \frac{n(n-1)}{2} + n$$

$$= n^2/2 - n/2 + n$$

$$= 1/2(n^2 + n)$$

$$= \Theta(n^2).$$

Only polynomial number of unique subproblems.

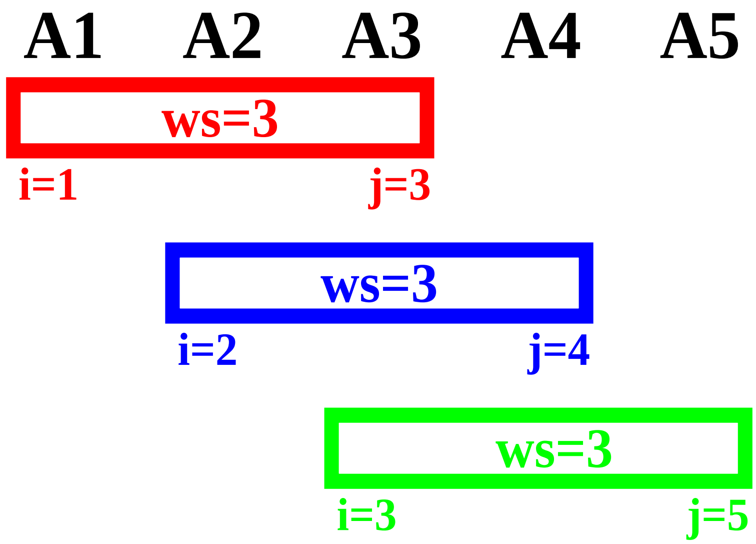
Step 3: Bottom-Up Approach

Compute optimal costs using a Bottom-Up approach.

If we solve smallest subproblems first, then larger problems will be easier to solve.

Define Arrays

- m[1..n, 1..n] for minimum costs
- s[1..n, 1..n] for optimal splits



Dynamic Programming

```
Matrix-Chain-Order(p)
    n = length(p) - 1
    for i = 1 to n
        m[i,i] = 0                    ; Chains of length 1
    for ws = 2 to n
        for i = 1 to n - (ws - 1)
            j = i + (ws - 1)
            m[i,j] = ∞
            for k = i to j-1
                q = m[i,k] + m[k+1, j] + P[i-1]P[k]P[j]
                if q < m[i,j]
```

```
                then m[i,j] = q
                     s[i,j] = k
     return m and s
```

This algorithm requires $\Theta(n^3)$ time and $\Theta(n^2)$ memory.

Step 4: Construct Optimal Solution

Let A = $\langle A_1, A_2, .., A_n \rangle$.

Call Matrix-Chain-Order then Matrix-Chain-Multiply, defined below.

```
Matrix-Chain-Multiply(A, s, i, j)
   if i < j
   then x = Matrix-Chain-Multiply(A, s, i, s[i,j])
        y = Matrix-Chain-Multiply(A, s, s[i,j]+1, j)
        return Matrix-Multiply(x, y)
   else return A_i
```

Elements of Dynamic Programming

1. ______________________________ Optimal solution to problem involves optimal solutions to subproblems.

2. __________________________________ Of the typically exponential number of subproblems referred to by a recursive solution, only a polynomial number of them are distinct.

Memoization

Top-Down recursive solution that remembers intermediate results.

For example, intermediate results found in m[2,4] are useful in determining the value of m[1,3].

```
Memoized-Matrix-Chain(p)
1    n = length(p) - 1
2    for i = 1 to n
3        for j = i to n
4            m[i,j] = ∞
5    return Lookup-Chain(p, 1, n)
Lookup-Chain(p, i, j)
1    if m[i,j] < ∞
2    then return m[i,j]
3    if i = j
4    then m[i,j] = 0
5    else for k = i to j-1
6            q = Lookup-Chain(p, i, k) +
                 Lookup-Chain(p, k+1, j) + P[i-1]P[k]P[j]
7            if q < m[i,j]
8            then m[i,j] = q
9    return m[i,j]
```

In this algorithm each of $\Theta(n^2)$ entries is initialized once (line 4) and is filled in by one call to Lookup-Chain.

Each of $\Theta(n^2)$ calls to Lookup-Chain takes n steps ignoring recursion, so the total time required is $\Theta(n^2) * O(n) = O(n^3)$.

The algorithm requires $\Theta(n^2)$ memory.

Longest Common Subsequence (LCS)

Problem: Given two sequences X = $\langle x_1, .., x_m \rangle$ and Y = $\langle y_1, .., y_n \rangle$, find the longest subsequence Z = $\langle z_1, .., z_k \rangle$ that is common to x and y.

A subsequence is a subset of elements from the sequence with strictly increasing order (not necessarily contiguous).

For example, if X = $\langle$A,B,C,B,D,A,B$\rangle$ and Y = $\langle$B,D,C,A,B,A$\rangle$, then some common subsequences are:

- $\langle$A$\rangle$
- $\langle$B$\rangle$
- $\langle$C$\rangle$
- $\langle$D$\rangle$
- $\langle$A,A$\rangle$
- $\langle$B,B$\rangle$
- $\langle$B,C,A$\rangle$
- $\langle$B,C,B,A$\rangle$ This is one of the longest common subsequences.
- $\langle$B,D,A,B$\rangle$ This is one of the longest common subsequences.

Brute Force: Check all 2^m subsequences of X for an occurrence in Y.

Dynamic Programming

1. Optimal Substructure.

Define: Given X = $\langle x_1, .., x_m \rangle$, the ith prefix of X, i = 0, .., m, is $X_i = \langle x_1, .., x_i \rangle$. X_0 is empty.

Theorem 16.1

Let X = $\langle x_1, .., x_m \rangle$ and Y = $\langle y_1, .., y_n \rangle$ be sequences, and Z = $\langle z_1, .., z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

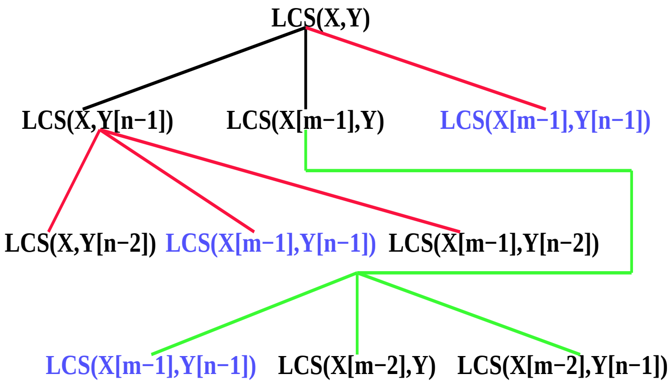
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Thus the LCS problem has optimal substructure.

Dynamic Programming

2. Overlapping Subproblems.



Define: c[i,j] = length of LCS for X_i and Y_j.

$$
\text{c[i,j]} = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}
$$

Distinct Subproblems

Could write an exponential recursive algorithm, but there are only ____ distinct subproblems.

Solution

Let c[i,j] be maximum length array.

Let b[i,j] record the case relating X_i, Y_j, and Z_k.

```
LCSLength(x, y)                       j <---
   m = length(x)                   +------------
   n = length(y)                   | \
   for i = 1 to m                i |  \
      c[i,0] = 0                 ^ |   \
   for j = 0 to n                | |
      c[0,j] = 0                 | |
   for i = 1 to m                | |
      for j = 1 to n               |
         if x[i] = y[j]
         then c[i,j] = c[i-1,j-1] + 1
              b[i,j] = '\'             ; Arrow points up and le
         else if c[i-1,j] >= c[i,j-1]
              then c[i,j] = c[i-1,j]
                 b[i,j] = '^'          ; Up arrow
```

```
          else c[i,j] = c[i,j-1]
             b[i,j] = '<'          ; Left arrow
return c and b
```

LCSLength is O(mn).

Pseudocode

```
PrintLCS(b, X, i, j)
   if i=0 or j=0
   then return
   if b[i,j] = '\'
   then PrintLCS(b, X, i-1, j-1)
        print x[i]
   else if b[i,j] = '^'
        then PrintLCS(b, X, i-1, j)
        else PrintLCS(b, X, i, j-1)
```

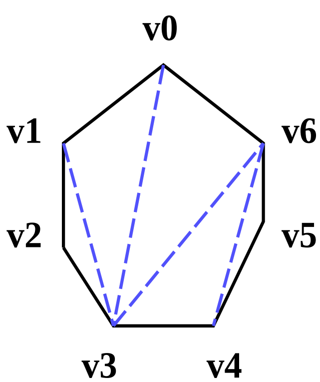
PrintLCS is O(m+n).

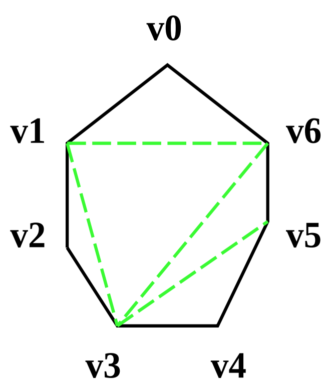
```
         0   1   2   3   4   5
       y[j]  b   r   o   w   n
       +---+---+---+---+---+---+   PrintLCS(b, "cow", 3, 5) <
0 x[i] | 0 | 0 | 0 | 0 | 0 | 0 |     PrintLCS(b, "cow", 3, 4) \
       +---+---+---+---+---+---+       PrintLCS(b, "cow", 2, 3)
1   c  | 0 | 0 | 0 | 0 | 0 | 0 |         PrintLCS(b, "cow", 1,
       +---+---+---+---+---+---+
2   o  | 0 | 0 | 0 |\1 |<1 |<1 |   o w
```

```
       +---+---+---+---+---+---+
3   w  | 0 | 0 | 0 |^1 |\2 |<2 |
       +---+---+---+---+---+---+
```

Optimal Polygon Triangulation

- A polygon is described by P $= \angle v_0, v_1, .., v_{n-1}\rangle$.

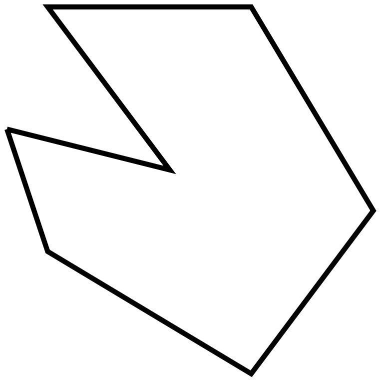




Optimal Polygon Triangulation

- A polygon is **convex** if the line segment between any two points lies on the boundary or the interior.

  This polygon is not convex.



- If v_i and v_j are not adjacent, segment $\overline{v_i v_j}$ is a _______.

- A __________________ is a set of chords T that divides P into disjoint triangles.
  - No chords intersect
  - T is maximal (every chord $\notin T$ intersects a cord $\in T$).

Optimal Polygon Triangulation

Problem:

- Given:
  - P = $\langle v_0, v_1, .., v_{n-1} \rangle$
  - A weight function w on triangles formed by P and T.
- Find T that minimizes the sum of weights

- Example: $w(\triangle v_i v_j v_k) = \mid v_i v_j \mid + \mid v_j v_k \mid + \mid v_k v_i \mid$ (Euclidean distance)
- Looks a bit like matrix chaining
- Optimal substructure
  - T contains $\triangle v_0 v_k v_n$.
    $w(T) = w(\triangle v_0 v_k v_n) + m[0, k] + m[k+1, n]$.
  - The two subproblem solutions must be __________ or ____________________
- This algorithm requires $\Theta(n^3)$ time.
- This algorithm requires $\Theta(n^2)$ memory.

Applications